National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2018/2019

Solutions for Recitation 4
Data Abstraction

Python

1. *Tuple* - `(`*value1*, *value2*, ...`)`

   A tuple is an immutable sequence of Python objects enclosed in parentheses and separated by commas.

2. Operations on tuples:
   (a) *len(x)* - Returns the number of elements of tuple x.
   (b) *element* in *x* - Returns `True` is *element* is in x, and `False` otherwise.
   (c) for *var* in *x* - Will iterate over all the elements of x with variable var.
   (d) *max(x)* - Returns the maximum element in the tuple x.
   (e) *min(x)* - Returns the miniimum element in the tuple x.

Problems

1. Evaluate the following expressions:

```
>>> tup_a = (10, 12, 13, 14) #Creating tup_a
>>> print(tup_a)
(10, 12, 13, 14)

>>> tup_b = ("CS1010S", "CS1231") #Creating tup_b
>>> print(tup_b)
('CS1010S', 'CS1231')

>>> tup_c = tup_a + tup_b #Creating tup_c
>>> print(tup_c)
(10, 12, 13, 14, 'CS1010S', 'CS1231')

>>> len(tup_c)
6

>>> 14 in tup_a
True

>>> 11 in tup_c
False
```

```
>>> tup_d = tup_b[0] * 4

>>> tup_d[0]
'C'

>>> tup_d[1:]
'S1010SCS1010SCS1010SCS1010S'
# Note that tup_d is actually a string and NOT a tuple!

>>> count = 0
>>> for i in tup_a:
    count = count + i
>>> print(count)
49

>>> max(tup_a)
14

>>> min(tup_a)
10

>>> max(tup_c)
# Error message, because string object
# cannot be compared with integer number

>>> min(tup_c)
# Error message, because string object
# cannot be compared with integer number
```

2. Write expressions whose values will print out like the following.

```
(1, 2, 3)
=> (1, 2, 3)
=> (1, ) + (2, ) + (3, )
=> (1, ) + (2, 3)
=> (1, 2) + (3, )

(1, (2), 3)
# Trick question -- cannot be done

(1, (2,), 3)
=> (1, ) + ((2, ),) + (3, )

((1, 2), (3, 4), (5, 6))
=> ((1, 2), ) + ((3, 4), (5, 6))
```

3. Write expressions to that will return the value 4 when the `x` is bound to the following values:

```
(7, 6, 5, 4, 3, 2, 1)
=> x[3]

(7, (6, 5, 4), (3, 2), 1)
=> x[1][2]

(7, ((6, 5, (4,), 3), 2), 1)
=> x[1][0][2][0]
```

4. You found a holiday assignment at the Registar's Office. Your job is to write a program to help students with their scheduling of classes. You are provided with an implementation of the records for each class as follows:

```
def make_module(course_code, units):
    return (course_code, units)

def make_units(lecture, tutorial, lab, homework, prep):
    return (lecture, tutorial, lab, homework, prep)

def get_module_code(course):
    return course[0]

def get_module_units(course):
    return course[1]

def get_module_total_units(units):
    return units[0] + units[1] + units[2] + units[3] + units[4]
```

Each class (course) has a course code and an associated number of credit unit, e.g. for CS1101S, that's 3-2-1-3-3. Your job is now to write a schedule object to represent a set of classes taken by a student. **Note:** Since `class` is a keyword in Python, we will use `course` as the variable representing the class currently of interest.

(a) Write a constructor `make_empty_schedule()` that returns an empty schedule.

```
def make_empty_schedule():
    return ()
```

Order of growth in time: $O(1)$, space: $O(1)$.

(b) Write a function `add_class` that when given a class and a schedule, returns a new schedule including the new class:

```
def add_class(course, schedule):
    if (course in schedule):
        return schedule
    else:
        return (course, ) + schedule
```

Order of growth in time: $O(n)$, space (stack): $O(1)$, space (heap): $O(n)$, where n is `len(schedule)`

It take $O(n)$ time for `course in schedule` to scan all the elements in `schedule` to check if `course` is equal to each element. It also takes $O(n)$ to create a new tuple.

In terms of space, `add_class` produces a tuple of size equal to the size of the `schedule` and so it takes $O(n)$ space also.

Alternative, recursive approach:

```
def add_class(course, schedule):
    if schedule == ():
        return (course, )
    elif course == schedule[0]:
        return schedule
    else:
        return (schedule[0],) + add_class(course, schedule[1:])
```

Order of growth in time: $O(n^2)$, space (stack): $O(n)$, space (heap): $O(n^2)$.

The concaternation of two tuples however takes time equal to the number of elements, so in `(schedule[0],) + add_class(course, schedule[1:])`, the worst case total time is $1 + 2 + \cdots + n = \frac{n}{2}(n+1) = O(n^2)$.

Every recursive call slices the tuple by one less element, which creates a new tuple of size $n - 1$. Thus, a new tuple is created for each recursive call, taking up space in the heap.

(c) Write a function `total_scheduled_units` that computes the total number of units in a specified schedule.

```
def get_units(course):
    return get_module_total_units(get_module_units(course))

def total_scheduled_units(schedule):
    total = 0
    for course in schedule:
        total = total + get_units(course)
    return total
```

Order of growth in time: $O(n)$, space: $O(1)$.

(d) Write a function `drop_class` that returns a new schedule with a particular class dropped from a specified schedule.

```
def drop_class(schedule, course):
    if schedule == ():
        return schedule
    elif schedule[0] == course:
        return drop_class(schedule[1:], course)
        # if courses in schedule are unique,
        # it is enough to just return schedule[1:]
    else:
        return (schedule[0], ) + drop_class(schedule[1:], course)
```

Order of growth in time: $O(n^2)$, space (stack): $O(n)$, space (heap): $O(n^2)$.

(e) Implement a credit limit by taking in a schedule, and returning a new schedule that has total number of units is less than or equal to `max_credits` by removing classes from the specified schedule.

```
def credit_limit(schedule, max_credits):
    total = total_scheduled_units(schedule)
    while (total > max_credits):
        total -= get_units(schedule[0])
        schedule = schedule[1:]
    return schedule
```

Order of growth in time: $O(n^2)$, space (stack): $O(1)$, space (heap): $O(n)$. It takes $O(n)$ time and $O(n)$ (heap) space for creating the slice `s[1:]` where n = `len(s)`.

(f) **Homework:** Implement an improved version of `credit_limit` that will return a schedule with a total number of units is less than or equal to `max_credits`, but with the maximal number of classes. What is the order of growth of your solution? Is that the best you can do?

```
def credit_limit(schedule, max_credits):
    total = total_scheduled_units(schedule)
    if (total <= max_credits):
        return schedule
    elif max_credits < 0:
        return None      # No solution
    else:
        # Find best schedule by dropping the first course
        schedule1 = credit_limit(schedule[1:], max_credits)

        # Find best schedule without dropping the first course
        new_max_credits = max_credits - get_units(schedule[0])
        schedule2 = credit_limit(schedule[1:], new_max_credits)

        # If schedule2 not viable, only alternative is schedule1
        # (may be None)
        if (schedule2 == None):
            return schedule1

        # Schedule2 is viable, need to add the first course back
        schedule2 = (schedule[0], ) + schedule2

        # If schedule1 not viable, only alternative is schedule2
        if (schedule1 == None):
            return schedule2

        # Both viable; choose the schedule with more courses
        if (len(schedule1) >= len(schedule2)):
            return schedule1
        else:
            return schedule2

    # IT IS SIMILAR TO THE COUNT CHANGE PROBLEM, ISN'T IT?
```

What is the order of growth of your solution: it is similar to `count change` problem. Please work it out. This is not the best we can do and we will visit similar problems in the future.